Using the Streaming SIMD Extensions 2 (SSE2) to Evaluate a Hidden Markov Model with Viterbi Decoding

Version 2.0

7/00

Order Number: 248671-001

Table of Contents

Revision History

Revision	Revision History	Date
2.0	Updated for the Pentium® 4 Processor	7/00
1.0	Original publication of document	9/99

References

The following documents are referenced in this application note, and provide background or supporting information for understanding the topics presented in this document.

1. L.R. Rabiner, "A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition," *Proceedings of the IEEE,* February 1989, Vol. 77, No. 2.
2. Richard A., Quinnel, "Speech Recognition: No Longer a Dream but Still a Challenge," *EDN*, January 19, 1995, pgs. 41 – 46.
3. *Using the Streaming SIMD Extensions to Evaluate a Hidden Markov Model with Viterbi Decoding*, Intel Application Note, AP-811, Copyright 1998 (See http://developer.intel.com/vtune/cbts/strmsimd/811down.htm)
4. *Using MMX™ Instructions to Implement Viterbi Decoding*, Intel Application Note, AP-569, Copyright 1996, http://developer.intel.com/drg/mmx/appnotes/ap569.htm.
5. *Dynamic Programming*, Ch 7, pgs 7-1 to 7-24, Intel's Recognition Primitives Library.
6. L.R. Rabiner and B. Juan, *Fundamentals of Speech Recognition*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1993, pg340

1 Introduction

The Streaming SIMD Extensions 2 (SSE2) technology introduces new Single Instruction Multiple Data (SIMD) double-precision floating-point instructions and new SIMD integer instructions into the IA-32 Intel® architecture. The double-precision SIMD instructions extend functionality in a manner analogous to the single-precision instructions introduced with the Streaming SIMD Extensions (SSE). The 128-bit SIMD integer extensions are a full superset of the 64-bit integer SIMD instructions, with additional instructions to support more integer data types, conversion between integer and floating-point data types, and efficient operations between the caches and system memory. These instructions provide a means to accelerate operations typical of 3D graphics, real-time physics, spatial (3D) audio, video encoding/decoding, encryption, and scientific applications. The 128-bit integer SIMD extensions in the SSE2 instructions can process data 128 bits at a time using the XMM registers, enabling the implementation of important algorithms, such as the Viterbi decoding algorithm and the evaluation of the Hidden Markov Model, to be improved further than previous implementations using MMX™ technology and SSE. This application note contains both the code and a description of how the SSE2 instructions can be used to implement a Viterbi algorithm to evaluate a Hidden Markov Model.

Application note, AP-569, entitled *Using MMX™ Instructions to Implement Viterbi Decoding*, describes how to use the MMX instructions to gain a 2x improvement over scalar code. Another application note, AP-811, entitled *Using the Streaming SIMD Extensions to Evaluate a Hidden Markov Model with Viterbi Decoding,* shows how using the SSE instructions and operating on four data elements at a time (increasing the SIMD width by two) can further increase the performance gain. This application note will describe how the SSE2 instructions provide a significant performance gain when compared to the implementation that uses the SSE instructions.

2 Viterbi Decoding Algorithm

This section was taken from section 2 of AP-811. This section briefly discusses how the Viterbi decoding algorithm is used to evaluate a Hidden Markov Model (HMM). The reader is encouraged to use the references listed earlier in this document to study these topics in more detail. Section 2.2 in AP-811 provides some background information on HMMs.

An HMM (λ) consists of a Markov chain of *n* states, numbered j = 1, 2, … , n [5]. Transitions can occur from one state (*i)* to another state (*j)*. An HMM is described by the parameter set {A, B, π} as follows:

- A is the matrix $\{a_{ij}\}$ called the transition probability, where a_{ij} is the probability of a transition from state *i* to state *j*.
- B is the matrix $\{b_i(k)\}$ called the output probability, where $b_i(k)$ is the probability of producing observation *k* when in state *i*.
- π is the vector $\{\pi_i\}$ called the initial probability, where π_i is the probability of starting in state *i*.

Different HMM structures can be used depending on the types of transitions that can occur. This document only considers the constrained-jump model. As shown in Figure 1, this model limits the state transitions. Transitions can only occur from state *i* to state *i*, *i+1*, or *i+2* ($a_{ij} = 0$ for $j > i+2$ and $j<i$).

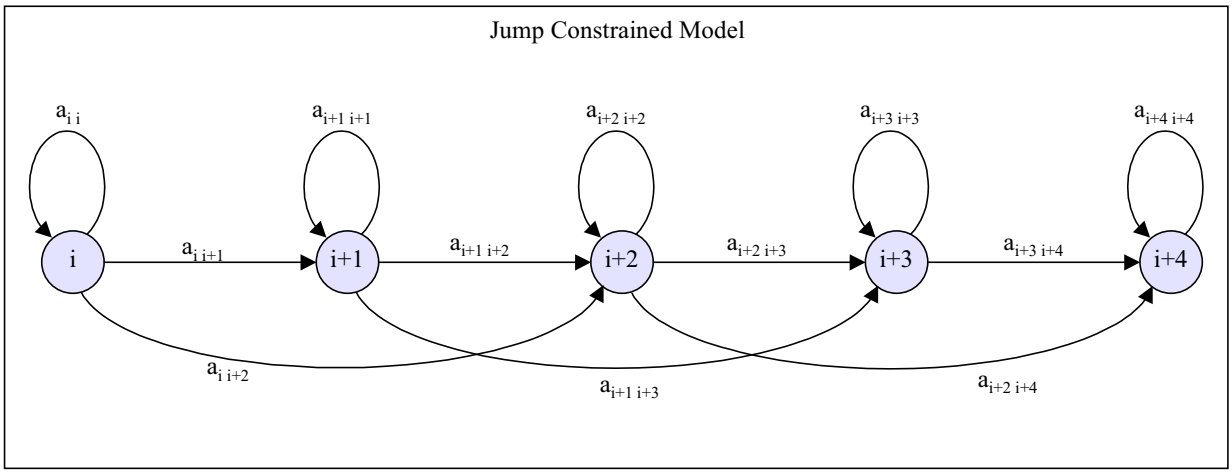

Figure 1: Jump Constrained Hidden Markov Model

In this application note only discrete HMMs are considered. In a discrete HMM a state can generate M distinct observation symbols. Each observation symbol corresponds to a particular physical output of the system being modeled. The individual observation symbols are denoted as $V = \{v_1, v_2, \ldots, v_M\}$. The indices of the observation symbols describe a sequence of physical outputs by means of an observation vector:

$O = \{O_1, O_2, \ldots, O_T\}$, where T is the number of observations in the sequence

Each individual observation in the observation vector is an index of an observation symbol that best matches the system's output. In a discrete HMM, the output probability $b_i(k)$ is the probability that the observation symbol v_k can be generated when in state *i* [1] (see Figure 2):

$$b_i(k) = P(v_k \text{ at time } t | q_t = \text{state } i), \text{ where } k = O_t,$$

$$1 \le i \le N,$$

$$1 \le k \le M$$

Given a discrete HMM with five states, each state has three probabilities associated to it:

π - Initial Probability
A - Transition Probability
B - Output Probability

Assuming this model has three observation symbols and is a jump constrained model, the probability matrices are as follows:

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & 0 & 0 \\ 0 & a_{22} & a_{23} & a_{24} & 0 \\ 0 & 0 & a_{33} & a_{34} & a_{35} \\ 0 & 0 & 0 & a_{44} & a_{45} \\ 0 & 0 & 0 & 0 & a_{55} \end{bmatrix} \quad B = \begin{bmatrix} b_1(1) & b_2(1) & b_3(1) & b_4(1) & b_5(1) \\ b_1(2) & b_2(2) & b_3(2) & b_4(2) & b_5(2) \\ b_1(3) & b_2(3) & b_3(3) & b_4(3) & b_5(3) \end{bmatrix} \quad \pi = \begin{bmatrix} \pi_1 \\ \pi_2 \\ \pi_3 \\ \pi_4 \\ \pi_5 \end{bmatrix}$$

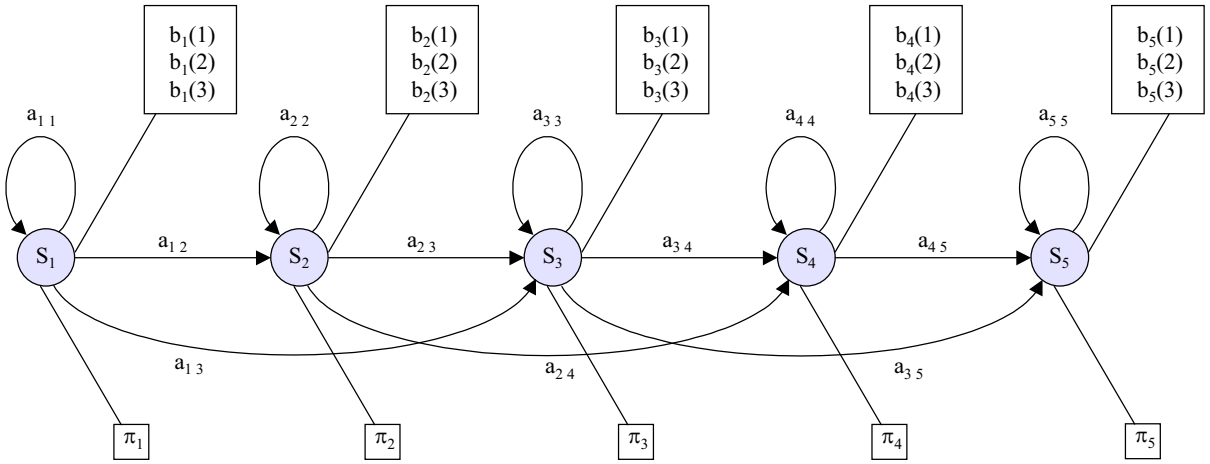

Figure 2: The parameters of a discrete Hidden Markov Model

HMMs can be used in speech recognition. A sequence of sound spectra generated by an utterance (words or syllables) can be modeled through a set of state transitions [2]. Given a speech sample represented by an observation vector O, one can compute the probability that the observation sequence can be generated by a given HMM, $P(O|\lambda)$. The model may generate the same observation sequence through different state sequences, but each state sequence is associated with a different probability. The state sequence with the best probability is the path that is most likely to generate the observation sequence. This state sequence is known as the best path (P^V):

$$P^V = \max[P(Q|O,\lambda)], \text{ where Q is a state sequence that can generate O}$$
$$P(O|\lambda) \approx P^V$$

To do an exhaustive search on an HMM for the best path is impractical. Instead, the Viterbi decoding algorithm provides an efficient method to find the best path. This algorithm can be used to find both the best path and the state sequence of the best path. Finding the state sequence of the best path is also known as backtracking [6]. The implementation in this application note only solves for the best path and does not solve for the backtrace.

The Viterbi algorithm is used to find the best path (P^V) given an observation sequence O and a HMM model λ. The description of the Viterbi algorithm follows [1]:

- Define $\delta_t(i)$ to be the best score (highest probability) along a single path at time *t*. This score considers the first t observations and the state sequence ends in state S_i:

$$\delta_t(i) = \max_{q_1, q_2, ..., q_{t-1}} P(q_1 q_2 ... q_t = S_i, O_1 O_2 ... O_t | \lambda)$$

And

$$\delta_{t+1}(j) = [\max_i \delta_t(i) a_{ij}] * b_j(O_{t+1})$$

- Viterbi Algorithm:

1) Initialization:

$$\delta_1(i) = \pi_i b_i(O_1) \qquad 1 \le i \le N \qquad (1)$$

2) Recursion:

$$\delta_t(j) = \max_{i = j, j-1, j-2} [\delta_{t-1}(i) a_{ij}] * b_j(O_t) \qquad 2 \le t \le T, \; 1 \le j \le N \qquad (2)$$

3) Termination:

$$P^V = \max_{1 \le i \le N} [\delta_T(i)] \qquad (3)$$

2.1 Applications for Evaluating HMMs using the Viterbi Decoding Algorithm

One application for the Hidden Markov Model is in an Isolated Word Recognizer (IWR)[1] (276-277). Rabiner discusses how each HMM is designed and trained for each word *W* in the vocabulary of the system. The IWR first performs feature analysis on an incoming speech signal – a spoken word (see Figure 3). The feature analysis converts a speech signal into a time sequence of coded spectral vectors (the observation sequence). As discussed above, each observation is an index of an observation symbol that best matches the physical output. In this speech recognition system, the observation symbols are coded spectral vectors. Once the observation sequence is derived from the input signal, the probability of generating the observation sequence given an HMM is computed for all models, $P(O|\lambda)$. To find $P(O|\lambda)$, each HMM is evaluated by the Viterbi algorithm to find the best path per HMM. The best path represents how well the most likely sequence of states in an HMM explains the given sequence of observations. The spoken word is considered recognized when the model with best probability is found:

$$w^* = \operatorname*{argmax}_{1 \le w \le W} [P(O|\lambda^W)]$$

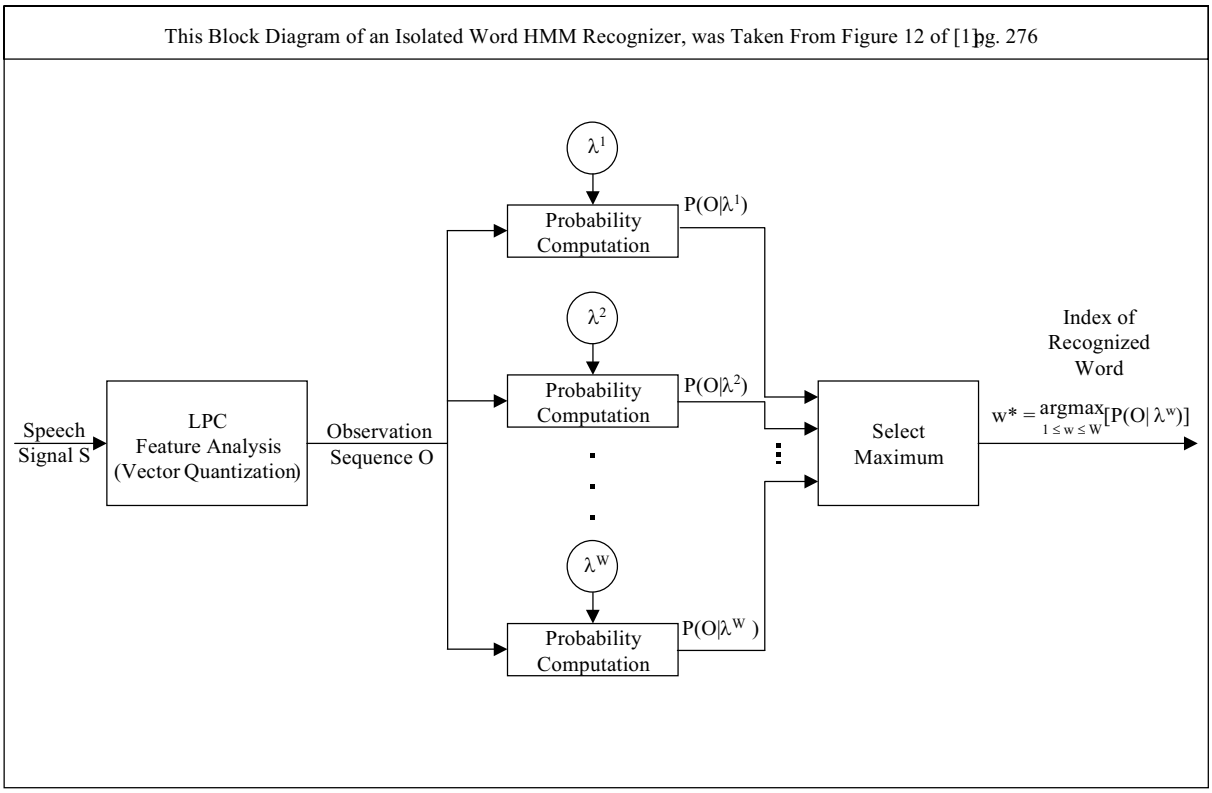

Figure 3: Block Diagram of Isolated Word Recognizer

2.2 Implementing the Viterbi Decoding Algorithm

The three equations that make up the Viterbi algorithm are implemented by taking the negative logarithms on each side. This is done to convert all multiplication operations into additions of positive numbers, in an attempt to avoid scaling issues. Using the previous application note's notation [3], [4], the three equations of the Viterbi algorithm are:

1) Initialization:

$$Dist(j,1) = Pi(j) + bProb(j,k(1)) \qquad 1 \le j \le N \qquad (4)$$

2) Recursion:

$$Dist(j,t+1) = \min_{i = j,\, j-1,\, j-2} [Dist(i,t) + aProb(j,i)] + bProb(j,k(t+1)) \qquad \begin{matrix} 1 \le t \le T \\ 1 \le j \le N \end{matrix} \qquad (5)$$

3) Termination:

$$Dist^{v} = \min_{1 \le j \le N} Dist(j,T) \qquad (6)$$

Where:

$Dist^{v} = -\log(P^{V})$
$Dist(j,t) = -\log(\delta_t(j))$
$aProb(j,i) = -\log(a_{ij})$
$bProb(j,k(t)) = -\log(b_j(O_t))$
$Pi(j) = -\log(\pi_j)$

Equation 5 is where most of the processing time of this algorithm is spent. This equation can be expanded as follows:

$$\text{Dist}(j,t+1) = \min\left[\{\text{Dist}(j,t) + \text{aProb}(j,j)\}, \{\text{Dist}(j-1,t) + \text{aProb}(j,j-1)\}, \{\text{Dist}(j-2,t) + \text{aProb}(j,j-2)\}\right] + \text{bProb}(j,k(t+1)) \quad \begin{matrix} 1 \le t \le T \\ 1 \le j \le N \end{matrix} \qquad (7)$$

This equation can be placed in SIMD form as demonstrated in Figure 4:

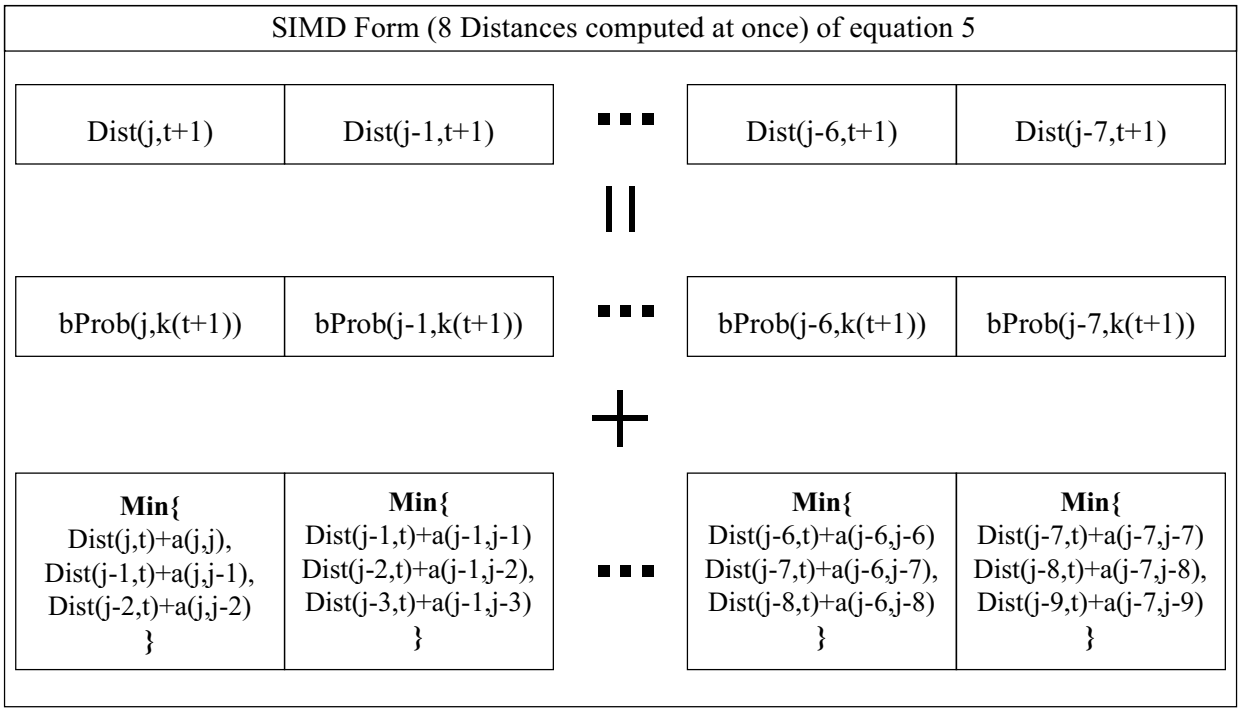

Figure 4: Equation 5 in SIMD Form.

Note: For simplicity, the aProb elements were shortened from aProb(x,y) to a(x,y). Also note that the left-most column is the scalar version of Eq 7.

In order to use the SIMD form of equation 5, the distances are computed by using decreasing values of j, (from j = N, N-1, ... , 1). The Intel® C++ SIMD Class Libraries for SIMD Operations can be used to express the SIMD form of Equation 5 with the SSE2 instructions:

```
for(...)
{
  //load variables dist8_1 and dist8_2
     sum0 = sat_add(dist8[j],a8[j]);
     sum1 = sat_add(dist8_1,a8_1[j]);
     sum2 = sat_add(dist8_2,a8_2[j]);

     min0 = simd_min(sum2,simd_min(sum1,sum0));
     dist8[j] = sat_add(b8[j],min0);
}
```

Even though step 2 of the Viterbi algorithm is where most of the processing time is spent, step 1 and step 3 of the algorithm can also be optimized through the use of SIMD instructions. See Section 6 for the Viterbi decoding algorithm implemented with the Intel C++ SIMD Class Libraries for SIMD Operations.

To fully understand the Viterbi decoding algorithm implementations in Sections 5-9, the reader needs to understand the parameters passed into the functions. The functions used in this application note need the following parameters:

1.	ObsVect	This is the observation vector that describes a speech signal as a sequence of observations, where each observation is an index that corresponds to an observation symbol.
2.	ObsLen	This is the length of the observation sequence (T).
3,4,5.	a, a_1, a_2	Three vectors are used to represent the transition probability matrix (aProb). The aProb matrix of a jump-constrained model only has three non-zero elements per row (see `vh 2). The three non-zero elements represent the three possible transitions per state j: aProb(j,j), aProb(j,j-1), and aProb(j,j-2). The vector "a" is used to represent the aProb(j,j) transition probability. The vector "a_1" is used to represent the aProb(j,j-1) transition probability. And finally, the vector "a_2" is used to represent the aProb(j,j-2) transition probability. These vectors are discussed in more detail below.
6.	bprob	This is the output probability, which is an array of pointers to arrays. The size of the array holding the pointers is also the number of observation symbols, M. The size of the array pointed by each pointer is the number of states, nStates (see Figure 5).
7.	Pi	This is the initial probability vector.
8.	nstates	This is the number of states in the HMM.
9.	DistBuffer	This is the temporary buffer that stores the accumulated distance quantities.

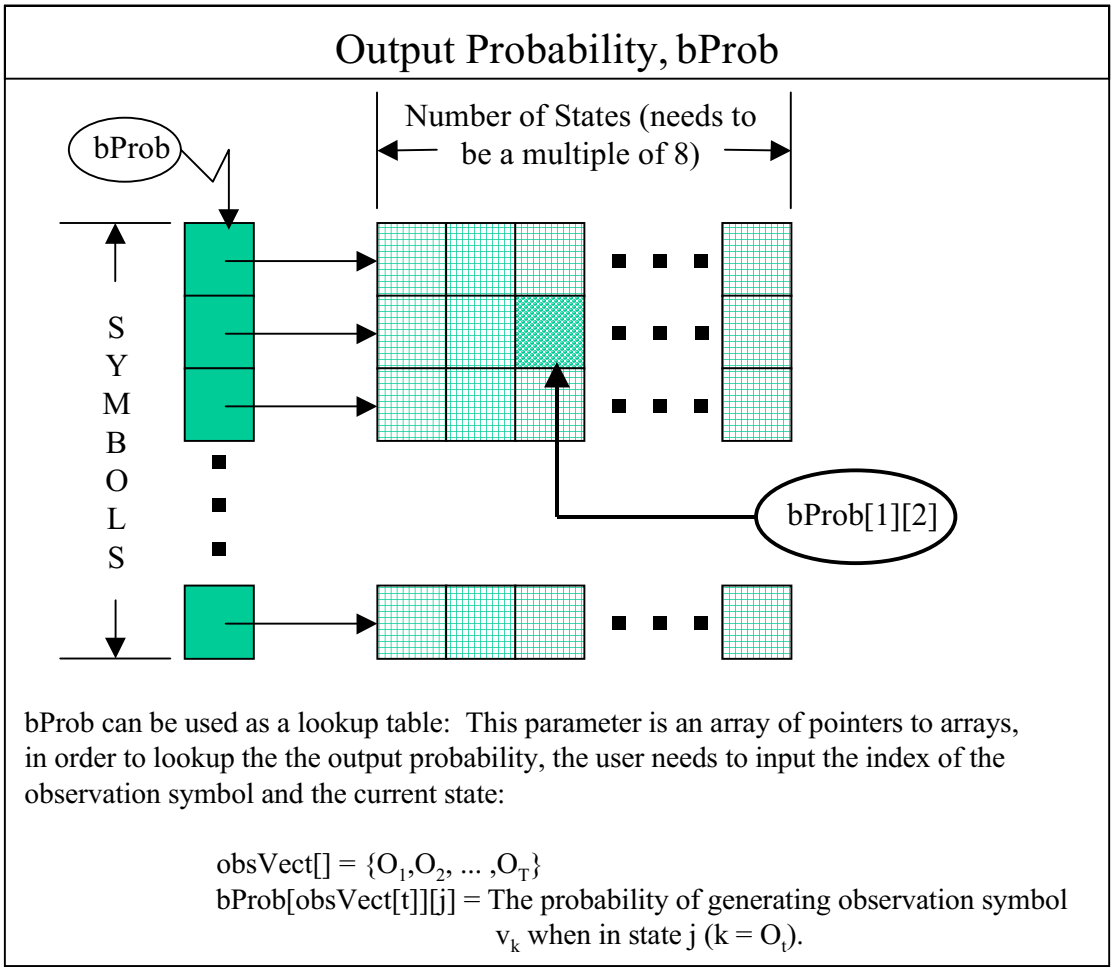

Figure 5: Pictorial description of the output probability (bProb) implementation

This application note implements the aProb matrix with three vectors, one for aProb(j,j), one for aProb(j,j-1), and one for aProb(j,j-2):

Vector a: a [] = {a(j,j), a(j-1,j-1), a(j-2,j-2), … ,a(j-n,j-n)}; //aProb(j,j)

Vector a_1: a_1[] = {a(j,j-1), a(j-1,j-2), a(j-2,j-3), … ,a(j-n,j-n-1)}; //aProb(j,j-1)

Vector a_2: a_2[] = {a(j,j-2), a(j-1,j-3), a(j-2,j-4), … ,a(j-n,j-n-2)}; //aProb(j,j-2)

The aProb matrix is implemented with three vectors to avoid unaligned loads. This implementation assumes the user has padded all the probability matrices/vectors to use a SIMD width of eight. That is, if an HMM has 6 states, it assumes that the programmer pads the distBuffer, Pi, aProb, and bProb arrays to a SIMD width of 8. This padding avoids the unnecessary branches and eliminates code to handle the extra elements.

2.2.1 Techniques

The SIMD implementation of Equation 5 requires at least two unaligned loads to process 8 distances per iteration. Assuming the distance vector is aligned on a 16-byte boundary, an aligned load (the `movdqa` instruction) can be used to load the first 8 distances. But the `movdqu` instruction needs to be used for the second and third unaligned loads:

```
movdqa xmm0, [edi+eax]    //Dist(j-7,t)...Dist(j,t)
movdqu xmm1, [edi+eax+2]  //Dist(j-8,t)...Dist(j-1,t), unaligned load
movdqu xmm2, [edi+eax+4]  //Dist(j-9,t)...Dist(j-2,t), unaligned load

paddsw xmm0, [ebx+eax]    //Dist8     += aProb8
paddsw xmm1, [ecx+eax]    //Dist8_1   += aProb8_1
paddsw xmm2, [edx+eax]    //Dist8_2   += aProb8_2

pminsw xmm0, xmm1
pminsw xmm0, xmm2         //Dist8=Min(Dist8,Dist8_1,Dist8_2)

paddsw xmm0, [esi+eax]    //Dist8 = Dist8 + bProb
movdqa [edi+eax], xmm0
```

The unaligned loads are a potential problem if the HMM contains a small number of states. That is, the data will *not* be forwarded when using an unaligned read immediately after the write. Instead, the unaligned read has to wait until the data has updated the memory hierarchy before the read can be executed.

One way to avoid the unaligned read stall (the store-forwarding penalty) is to keep the distances in registers. Another way to avoid an unaligned read is do two aligned reads and combine the data through shifts and an OR operation. These techniques can be implemented with inline ASM, using the Intel® C/C++ Compiler intrinsics, and/or the Intel C/C++ Class Libraries for SIMD Operations. These optimizations are demonstrated with different code samples described below:

1. **Functional implementation:**
   This is a functional, non-optimized implementation written in C++ that can be used to understand the Viterbi algorithm.
2. **Pentium III SIMD class implementation with general optimizations (PIVEC_GEN):**
   PIVEC_GEN uses the SIMD integer vector class (IVEC) of the Intel C/C++ Class Libraries for SIMD Operations to implement the Viterbi algorithm. Since this implementation is intended for the Pentium III processor, the `Is16vec4` (16-bit signed integer vector class of 4 elements) class is used. This class uses the 64-bit MMX technology registers and the MMX technology instructions and SSE instructions. This implementation substitutes the unaligned reads with two aligned reads, followed by two shifts, and one OR operation. This example demonstrates a general optimization (independent of the number HMM states). The aligned loads avoid the forwarding penalty.

3. **Pentium III SIMD class implementation with state-specific optimizations (PIVEC):** PIVEC is another implementation that uses the `Is16vec4` class to implement the Viterbi algorithm. This implementation also avoids the forwarding penalty and implements further optimizations that are specific to the number of HMM states. The comments in the code describe the optimizations used – see section 7.

4. **Pentium 4 SIMD class implementation with general optimizations (WIVEC_GEN):** WIVEC_GEN also uses the Intel C/C++ Class Libraries for SIMD Operations, but uses the `Is16vec8` class (16-bit signed integer vector class of 8 elements). This implementation is intended for the Pentium 4 processor. The `Is16vec8` class uses the XMM registers (the 128-bit registers) and the SSE2 instructions. This implementation uses the same general optimization as the PIVEC_GEN implementation, it avoids the store-forwarding stall by substituting each unaligned load with two aligned loads, two shifts, and one OR operation.

5. **Pentium 4 SIMD class implementation with state-specific optimizations (WIVEC):** WIVEC also uses the `Is16vec8` vector class and has been optimized to avoid the store-forwarding penalty and also includes other optimizations that are specific to the number of HMM states. The comments in the code describe the optimizations used – see section 9. Of the five implementations, this is the most efficient implementation discussed in this app note. This Pentium 4 processor IVEC implementation provides significant speedups compared to the best Pentium III processor IVEC implementation.

3 Performance

3.1 Gains/Improvements

The Viterbi decoding algorithm was previously described in application notes AP-569 and AP-811. The SSE2 instructions are used to further optimize the implementations in the aforementioned documents. The SSE2 instructions provide an increased SIMD width (processing of eight data elements instead of four data elements simultaneously). The increased SIMD width is the largest factor in the SSE2 instructions speedup. The `pminsw` instruction is also very instrumental in providing a speedup over the scalar code. The `pminsw` instruction is also used in the SSE implementation in AP-811.

Avoiding the store-forwarding penalty provided another significant speedup (see section 2.3.1). To avoid further unaligned loads, the state transition probabilities are also placed in three separate vectors. Padding the probability vectors and matrices provided another speedup by removing the unnecessary code and branches to handle the extra elements. Further optimizations can be made that are specific to the number of HMM states – see sections 7 and 9.

4 Conclusion

The SSE2 instructions can provide a significant performance gain in evaluating HMMs with the Viterbi algorithm. The increased SIMD width and the `pminsw` instruction are the largest factors in the speedup. Other optimizations include: avoiding the store-forwarding penalty, padding the vectors to avoid adding branches to handle the extra elements, and using optimizations that are specific to the number of HMM states.

5 Functional Code Example

This is a functional, non-optimized implementation written in C++ that can be used to understand the Viterbi algorithm.

```
short const MaxShort   = 0x7FFF;
//Signed 16-bit integers are used.  Thus positive numbers are saturated to
//0x7FFF
inline short add_with_saturate(short x, short y)
{
   return(((int)x+(int)y > MaxShort) ? MaxShort : (x+y));
}

inline short minx(short a, short b)
{
   return((a<b)?a:b);
}

//functional code
short hmm_kernel::functional(unsigned int *obsVect, int obsLen,
   short *a,short *a_1,short *a_2, short **bProb, short *pi, int nStates,
   short *distBuffer)
{
   short     minDist;
   short     *b = bProb[obsVect[0]];
   short     *dist = distBuffer;
   short     *dist_1 = &distBuffer[1];
   short     *dist_2 = &distBuffer[2];
   short     sum_0;
   short     sum_1;
   short     sum_2;
   short     min0;

   int       i = 0;

   if (nStates == 0)
     return(0);
```

```
  //step 1:  Initialization
  for( i = 0; i < nStates; i++)
    {dist[i] = add_with_saturate(pi[i],b[i]);}

  //Step 2:  Recursion
  /Loop for each observation or obsCount
  for(i = 1; i < obsLen; i++)
  {
    b = bProb[obsVect[i]];

    //OneStateLoop
    for(int j = 0; j < nStates; j++)
    {
      sum_0 = add_with_saturate(dist[j+0],a[j]);
      sum_1 = add_with_saturate(dist_1[j+0],a_1[j]);
      sum_2 = add_with_saturate(dist_2[j+0],a_2[j]);

      min0 = minx(sum_2,minx(sum_1,sum_0));
      dist[j+0] = add_with_saturate(b[j+0],min0);

    }

  }

  //Step 3:  Termination
  // Get Minimum Distance
  minDist = distBuffer[0];
  for (i = 1; i<nStates; i++)
  {
    minDist = minx(minDist,distBuffer[i]);
  }

  return(minDist);
}
```

6 Pentium III SIMD Implementation with General Optimization

PIVEC_GEN uses the SIMD integer vector class (IVEC) of the Intel C/C++ Class Libraries for SIMD Operations to implement the Viterbi algorithm. Since this implementation is intended for the Pentium III processor, the `Is16vec4` (16-bit signed integer vector class of 4 elements) class is used. This class uses the 64-bit MMX technology registers and the MMX technology instructions and SSE instructions. This implementation substitutes the unaligned reads with two aligned reads, followed by two shifts, and one OR operation. This example demonstrates a general optimization (independent of the number HMM states). The aligned loads avoid the forwarding penalty.

```
//Pentium(R) III processor - Ivec implementation - Uses the Streaming
//SIMD Extensions (Generic Optimization-avoids the store-forwarding penalty)
short hmm_pivec_gen::TheCode(unsigned int *obsVect, int obsLen, short *a,
               short *a_1,short *a_2, short **bProb, short *pi,
               int nStates, short *distBuffer)


{


  short     minDist;
  int i = 0;

  Is16vec4 *b4 = (Is16vec4 *)(bProb[obsVect[0]]);
  Is16vec4 *dist4   = (Is16vec4 *) distBuffer;
  Is16vec4 d4, d4next, d4_1, d4_2;
  Is16vec4 *a4   = (Is16vec4 *) a;
  Is16vec4 *a4_1 = (Is16vec4 *) a_1;
  Is16vec4 *a4_2 = (Is16vec4 *) a_2;
  Is16vec4 sum0, sum1, sum2, min0, min1;
  Is16vec4 *pi4 = (Is16vec4 *) pi;


  if (nStates == 0)
    return(0);

  int Pack4States = nStates >> 2;
```

```
////ERROR IF LESS THAN 4 STATES OR IF STATES ARE NOT A MULTIPLE OF 4////
  //assert(Pack4States == 0 || (nStates % 4) > 1)
  if (Pack4States == 0 || (nStates % 4) > 1)
    return(-1);
//////////////////////////////////////////////////////////////////////

  //Step 1 of Viterbi Algorithm:  Initialization
  for( i = 0; i < Pack4States; i++)
    {dist4[i] = sat_add(pi4[i],b4[i]);}

  //Loop for each observation or obsCount
  for(i = 1; i < obsLen; i++)
  {
    b4         = (Is16vec4 *)(bProb[obsVect[i]]);

    //Step 2 of Viterbi Algorithm:  Recursion
    //FourStateLoop
    for(int j = 0; j < Pack4States; j++)
    {
      d4     = dist4[j];
      d4next = dist4[j+1];

      d4_1 =  ((I64vec1)d4 >> 16);
      d4_1 |= ((I64vec1)d4next << 48);
      d4_2 =  ((I64vec1)d4 >> 32);
      d4_2 |= ((I64vec1)d4next << 32);


      sum0 = sat_add(d4,  a4[j]);
      sum1 = sat_add(d4_1,a4_1[j]);
      sum2 = sat_add(d4_2,a4_2[j]);

      min0 = simd_min(sum2,simd_min(sum1,sum0));
      dist4[j] = sat_add(b4[j],min0);

    }
```

```
  }
  //Step 3 of Viterbi Algorithm:  Termination
  // Get 4 Minimum Distances
  min0 = dist4[0];
  for(i = 1; i < Pack4States; i++)
  {
    min0 = simd_min(min0, dist4[i]);
  }

  // Get Minimum Distance
                                              //min0 = m3 m2 m1 m0
  min1 = (Is16vec4)_m_pshufw(min0, 0x0E);     //min1 = xx xx m3 m2
  min0 = simd_min(min0, min1);                //min0 = xx xx m1 m0
  min1 = (Is16vec4)_m_pshufw(min0, 0x01);     //min1 = xx xx xx m1
  min0 = simd_min(min0, min1);                //min0 = xx xx xx min
  minDist = _m_pextrw(min0, 0);               //minDist = min

  _m_empty();
  return(minDist);
}
```

7 Pentium III SIMD Implementation with State-Specific Optimizations

PIVEC is another implementation that uses the `Is16vec4` class to implement the Viterbi algorithm. This implementation also avoids the forwarding penalty and implements further optimizations that are specific to the number of HMM states.

```
//Pentium(R) III processor - Ivec implementation - Uses the Streaming
//SIMD Extensions (State-Specific Optimizations)
//For comparison with the WIVEC implementation, this implementation is
//only optimized for states that are a multiple of 8

short hmm_pivec::TheCode(unsigned int *obsVect, int obsLen, short *a,
               short *a_1,short *a_2, short **bProb, short *pi,
               int nStates, short *distBuffer)
{
  short     minDist;
  int i = 0;

  if (nStates == 0)
    return(0);

  int Pack4States = nStates >> 2;

////ERROR IF LESS THAN 4 STATES OR IF STATES ARE NOT A MULTIPLE OF 4////
  //assert(Pack4States == 0 || (nStates % 4) > 1)
  if (Pack4States == 0 || (nStates % 4) > 1)
    return(-1);
//////////////////////////////////////////////////////////////////////



  Is16vec4 *b4 = (Is16vec4 *)(bProb[obsVect[0]]);
  Is16vec4 *dist4   = (Is16vec4 *) distBuffer;
  Is16vec4 d4, d4next, d4_1, d4_2;
  Is16vec4 d4_1st, d4_1_1st, d4_2_1st;
  Is16vec4 d4_2nd, d4_1_2nd, d4_2_2nd;
  Is16vec4 *a4   = (Is16vec4 *) a;
  Is16vec4 *a4_1 = (Is16vec4 *) a_1;
```

```
Is16vec4 *a4_2 = (Is16vec4 *) a_2;
Is16vec4 sum0, sum1, sum2, min0, min1;
Is16vec4 sum0_1st, sum1_1st, sum2_1st, min0_1st, min1_1st;
Is16vec4 sum0_2nd, sum1_2nd, sum2_2nd, min0_2nd, min1_2nd;

Is16vec4 *pi4 = (Is16vec4 *) pi;

//For comparison with the WIVEC implementation, this implementation is
//only optimized for states that are a multiple of 8
switch (Pack4States)
{
case 2: //8 States
  {
    //step1:  Initialization of 8 states
    //Optimization:  Do two stores instead of a loop
    d4_1st = sat_add(pi4[0], b4[0]);
    d4_2nd = sat_add(pi4[1], b4[1]);

    //step2:  Loop
    //Optimization:  Try to keep the data in the registers by:
    //Optimization1:  Use only one loop instead of outer & inner loop.
    //Optimization2:  There are only 8 states, thus combining the data
    //                 with the third load is unnecessary and has been
    //                 commented out.
    for( i=1; i < obsLen; i++ )
    {
      b4 = (Is16vec4 *)(bProb[obsVect[i]]);

      ////////////// Do the 1st Packed 4 Distance ////////
      d4_1_1st  =  ((I64vec1)d4_1st >> 16);
      d4_1_1st |=  ((I64vec1)d4_2nd << 48);
      d4_2_1st  =  ((I64vec1)d4_1st >> 32);
      d4_2_1st |=  ((I64vec1)d4_2nd << 32);

      sum0_1st = sat_add(d4_1st,  a4[0]);
      sum1_1st = sat_add(d4_1_1st,a4_1[0]);
      sum2_1st = sat_add(d4_2_1st,a4_2[0]);
```

```
        min0_1st = simd_min(sum2_1st,simd_min(sum1_1st,sum0_1st));
        //store result of first packed 4 distance
        d4_1st = sat_add(b4[0],min0_1st);

        /////////////// Do the 2nd Packed 4 Distance /////////
        d4_1_2nd  =  ((I64vec1)d4_2nd     >> 16);
        //d4_1_2nd |=  ((I64vec1)d4_3rd << 48); - not needed in this case
        d4_2_2nd  =  ((I64vec1)d4_2nd     >> 32);
        //d4_2_2nd |= ((I64vec1)d4_3rd << 32); - not needed in this case

        sum0_2nd = sat_add(d4_2nd,  a4[1]);
        sum1_2nd = sat_add(d4_1_2nd,a4_1[1]);
        sum2_2nd = sat_add(d4_2_2nd,a4_2[1]);

        min0_2nd = simd_min(sum2_2nd,simd_min(sum1_2nd,sum0_2nd));
        //store result of second packed 4 distance
        d4_2nd = sat_add(b4[1],min0_2nd);
      }


      //optimization:  No need to use a loop
      //Step 3 of Viterbi Algorithm:  Termination
      // Get 4 Minimum Distances
      min0 = simd_min(d4_1st, d4_2nd);
      break;
    }

case 4: //16 States
    {

      Is16vec4 sum0_3rd, sum1_3rd, sum2_3rd, min0_3rd, min1_3rd;
      Is16vec4 sum0_4th, sum1_4th, sum2_4th, min0_4th, min1_4th;
      Is16vec4 d4_3rd, d4_1_3rd, d4_2_3rd;
      Is16vec4 d4_4th, d4_1_4th, d4_2_4th;

      //step1:  Initialization of 16 states
      //optimization:  No need to use a loop
      d4_1st = sat_add(pi4[0], b4[0]);
```

```
d4_2nd = sat_add(pi4[1], b4[1]);
d4_3rd = sat_add(pi4[2], b4[2]);
d4_4th = sat_add(pi4[3], b4[3]);


//step2:  Loop
//Optimization1:  Use only one loop instead of outer & inner loop.
//Optimization2:  There are only 16 states, thus combining the data
//                 with the fifth load is unnecessary and has been
//                 commented out.
for( i=1; i < obsLen; i++ )
{
  b4 = (Is16vec4 *)(bProb[obsVect[i]]);

  ////////////// Do the 1st Packed 4 Distance ////////
  d4_1_1st  =  ((I64vec1)d4_1st >> 16);
  d4_1_1st |=  ((I64vec1)d4_2nd << 48);
  d4_2_1st  =  ((I64vec1)d4_1st >> 32);
  d4_2_1st |=  ((I64vec1)d4_2nd << 32);

  sum0_1st = sat_add(d4_1st,  a4[0]);
  sum1_1st = sat_add(d4_1_1st,a4_1[0]);
  sum2_1st = sat_add(d4_2_1st,a4_2[0]);

  min0_1st = simd_min(sum2_1st,simd_min(sum1_1st,sum0_1st));
  //store result of first packed 4 distance
  d4_1st = sat_add(b4[0],min0_1st);

  ////////////// Do the 2nd Packed 4 Distance ////////
  d4_1_2nd  =  ((I64vec1)d4_2nd >> 16);
  d4_1_2nd |=  ((I64vec1)d4_3rd << 48);
  d4_2_2nd  =  ((I64vec1)d4_2nd >> 32);
  d4_2_2nd |= ((I64vec1)d4_3rd  << 32);

  sum0_2nd = sat_add(d4_2nd,  a4[1]);
  sum1_2nd = sat_add(d4_1_2nd,a4_1[1]);
  sum2_2nd = sat_add(d4_2_2nd,a4_2[1]);
```

```
  min0_2nd = simd_min(sum2_2nd,simd_min(sum1_2nd,sum0_2nd));
  //store result of second packed 4 distance
  d4_2nd = sat_add(b4[1],min0_2nd);

  ////////////// Do the 3rd Packed 4 Distance /////////
  d4_1_3rd  =  ((I64vec1)d4_3rd >> 16);
  d4_1_3rd |=  ((I64vec1)d4_4th << 48);
  d4_2_3rd  =  ((I64vec1)d4_3rd >> 32);
  d4_2_3rd |=  ((I64vec1)d4_4th << 32);

  sum0_3rd = sat_add(d4_3rd,  a4[2]);
  sum1_3rd = sat_add(d4_1_3rd,a4_1[2]);
  sum2_3rd = sat_add(d4_2_3rd,a4_2[2]);

  min0_3rd = simd_min(sum2_3rd,simd_min(sum1_3rd,sum0_3rd));
  //store result of first packed 4 distance
  d4_3rd = sat_add(b4[2],min0_3rd);

  ////////////// Do the 4th Packed 4 Distance /////////
  d4_1_4th  =  ((I64vec1)d4_4th     >> 16);
  //d4_1_4th |=  ((I64vec1)d4_5th << 48); - not needed in this case
  d4_2_4th  =  ((I64vec1)d4_4th     >> 32);
  //d4_2_4th |= ((I64vec1)d4_5th << 32); - not needed in this case

  sum0_4th = sat_add(d4_4th,  a4[3]);
  sum1_4th = sat_add(d4_1_4th,a4_1[3]);
  sum2_4th = sat_add(d4_2_4th,a4_2[3]);

  min0_4th = simd_min(sum2_4th,simd_min(sum1_4th,sum0_4th));
  //store result of second packed 4 distance
  d4_4th = sat_add(b4[3],min0_4th);

}



//Step 3 of Viterbi Algorithm:  Termination
// Get 4 Minimum Distances
```

```
      //optimization:  No need to use a loop
      min0 = simd_min(d4_1st, d4_2nd);
      min0 = simd_min(min0, d4_3rd);
      min0 = simd_min(min0, d4_4th);
      break;
    }
default:
  {

    //Step 1 of Viterbi Algorithm:  Initialization
    for( i = 0; i < Pack4States; i++)
      {dist4[i] = sat_add(pi4[i],b4[i]);}

    //Loop for each observation or obsCount
    for(i = 1; i < obsLen; i++)
    {
      b4         = (Is16vec4 *)(bProb[obsVect[i]]);

      //Step 2 of Viterbi Algorithm:  Recursion
      //FourStateLoop
      for(int j = 0; j < Pack4States; j++)
      {
        d4     = dist4[j];
        d4next = dist4[j+1];

        d4_1 =  ((I64vec1)d4 >> 16);
        d4_1 |= ((I64vec1)d4next << 48);
        d4_2 =  ((I64vec1)d4 >> 32);
        d4_2 |= ((I64vec1)d4next << 32);


        sum0 = sat_add(d4,  a4[j]);
        sum1 = sat_add(d4_1,a4_1[j]);
        sum2 = sat_add(d4_2,a4_2[j]);

        min0 = simd_min(sum2,simd_min(sum1,sum0));
        dist4[j] = sat_add(b4[j],min0);
```

```
          }

        }
        //Step 3 of Viterbi Algorithm:  Termination
        // Get 4 Minimum Distances
        min0 = dist4[0];
        for(i = 1; i < Pack4States; i++)
        {
          min0 = simd_min(min0, dist4[i]);
        }
        break;
      }
    }
    // Get Minimum Distance
                                                  //min0 = m3 m2 m1 m0
    min1 = (Is16vec4)_m_pshufw(min0, 0x0E);       //min1 = xx xx m3 m2
    min0 = simd_min(min0, min1);                  //min0 = xx xx m1 m0
    min1 = (Is16vec4)_m_pshufw(min0, 0x01);       //min1 = xx xx xx m1
    min0 = simd_min(min0, min1);                  //min0 = xx xx xx min
    minDist = _m_pextrw(min0, 0);                 //minDist = min

    _m_empty();
    return(minDist);
}
```

8 Pentium 4 SIMD Implementation with General Optimization

WIVEC_GEN also uses the Intel C/C++ Class Libraries for SIMD Operations, but uses the `Is16vec8` class (16-bit signed integer vector class of 8 elements). This implementation is intended for the Pentium 4 processor. The `Is16vec8` class uses the XMM registers (the 128-bit registers) and the SSE2 instructions. This implementation uses the same general optimization as the PIVEC_GEN implementation: it avoids the store-forwarding stall by substituting each unaligned load with two aligned loads, two shifts, and one OR operation.

```
//Streaming SIMD Extensions 2 (SSE2) - Ivec implementation (Generic
Optimization)
short hmm_wivec_gen::TheCode(unsigned int *obsVect, int obsLen, short *a,
               short *a_1,short *a_2, short **bProb, short *pi,
               int nStates, short *distBufferl)
{


  short     minDist;
  int i = 0;

  Is16vec8 *b8 = (Is16vec8 *)(bProb[obsVect[0]]);
  Is16vec8 *dist8   = (Is16vec8 *) distBuffer;
  Is16vec8 d8, d8next, d8_1, d8_2;

  //transition probability:
  //  a8   - the prob. of moving back to the same state
  //  a8_1 - the prob. of moving to the next state (to state j-1)
  //  a8_2 - the prob. of moving to state j-2
  Is16vec8 *a8   = (Is16vec8 *) a;
  Is16vec8 *a8_1 = (Is16vec8 *) a_1;
  Is16vec8 *a8_2 = (Is16vec8 *) a_2;

  Is16vec8 sum0, sum1, sum2, min0, min1;
  Is16vec8 *pi8 = (Is16vec8 *) pi;



  if (nStates == 0)
    return(0);
```

```
  int Pack8States = nStates >> 3;

////ERROR IF LESS THAN 8 STATES OR IF STATES ARE NOT A MULTIPLE OF 8////
  //assert(Pack8States == 0 || (nStates % 8) > 1)
  if (Pack8States == 0 || (nStates % 8) > 1)
    return(-1);
////////////////////////////////////////////////////////////////////////


  //Step 1 of Viterbi Algorithm:  Initialization
  for( i = 0; i < Pack8States; i++)
    {dist8[i] = sat_add(pi8[i],b8[i]);}

  //Step 2 of Viterbi Algorithm:  Recursion
  //Loop for each observation or obsCount
  for(i = 1; i < obsLen; i++)
  {
    b8         = (Is16vec8 *)(bProb[obsVect[i]]);

    //EightStateLoop
    for(int j = 0; j < Pack8States; j++)
    {
      d8 = dist8[j];
      d8next = dist8[j+1];

      //Remove the unaligned loads by combining two aligned loads together
      d8_1 =  (Is16vec8)_mm_srli_si128((I128vec1)d8, 2);
      d8_1 |= (Is16vec8)_mm_slli_si128((I128vec1)d8next,14);
      d8_2 =  (Is16vec8)_mm_srli_si128((I128vec1)d8 ,4);
      d8_2 |= (Is16vec8)_mm_slli_si128((I128vec1)d8next,12);

      sum0 = sat_add(d8,  a8[j]);
      sum1 = sat_add(d8_1,a8_1[j]);
      sum2 = sat_add(d8_2,a8_2[j]);

      min0 = simd_min(sum2,simd_min(sum1,sum0));
      dist8[j] = sat_add(b8[j],min0);
```

```
    }

  }

  //Step 3 of Viterbi Algorithm:  Termination
  // Get 8 Minimum Distances
  min0 = dist8[0];
  for(i = 1; i < Pack8States; i++)
  {
    min0 = simd_min(min0, dist8[i]);
  }

  // Get Minimum Distance
  //min0 = m7 m6 m5 m4 m3 m2 m1 m0
  //min1 = xx xx xx xx m7 m6 m5 m4
  //min0 = xx xx xx xx m3 m2 m1 m0
  min1 = (Is16vec8) _mm_shuffle_epi32(min0, 0xE);
  min0 = simd_min(min0, min1);
  //min0 = xx xx xx xx m3 m2 m1 m0
  //min1 = xx xx xx xx xx xx m3 m2
  //min0 = xx xx xx xx xx xx m1 m0
  min1 = (Is16vec8)_mm_shufflelo_epi16(min0, 0xE);
  min0 = simd_min(min0, min1);
  //min0 = xx xx xx xx xx xx m1 m0
  //min1 = xx xx xx xx xx xx xx m1
  //min0 = xx xx xx xx xx xx xx min
  min1 = (Is16vec8)_mm_shufflelo_epi16(min0, 0x1);
  min0 = simd_min(min0, min1);
  //minDist = min
  minDist = _mm_extract_epi16(min0, 0);

  return(minDist);
}
```

9 Pentium 4 SIMD Implementation with State-Specific Optimizations

WIVEC also uses the `Is16vec8` vector class and has been optimized to avoid the store-forwarding penalty and also includes other optimizations that are specific to the number of HMM states. The comments in the code describe the optimizations used. Of the five implementations, this is the most efficient implementation discussed in this app note. This Pentium 4 processor IVEC implementation provides significant speedups compared to the best Pentium III processor IVEC implementation.

```
//Streaming SIMD Extensions 2 (SSE2) - Ivec implementation (State-Specific
//Optimizations)
short hmm_wivec::TheCode(unsigned int *obsVect, int obsLen, short *a,
               short *a_1,short *a_2, short **bProb, short *pi,
               int nStates, short *distBufferl)
{


  short     minDist;
  int i = 0;

  Is16vec8 *b8 = (Is16vec8 *)(bProb[obsVect[0]]);
  Is16vec8 *dist8   = (Is16vec8 *) distBuffer;
  Is16vec8 d8, d8next, d8_1, d8_2;

  Is16vec8 d8_1st, d8_1_1st, d8_2_1st;
  Is16vec8 sum0_1st, sum1_1st, sum2_1st, min0_1st, min1_1st;
  Is16vec8 d8_2nd, d8_1_2nd, d8_2_2nd;
  Is16vec8 sum0_2nd, sum1_2nd, sum2_2nd, min0_2nd, min1_2nd;
  Is16vec8 d8_3rd, d8_1_3rd, d8_2_3rd;
  Is16vec8 sum0_3rd, sum1_3rd, sum2_3rd, min0_3rd, min1_3rd;



  //transition probability:
  //  a8   - the prob. of moving back to the same state
  //  a8_1 - the prob. of moving to the next state (to state j-1)
  //  a8_2 - the prob. of moving to state j-2
  Is16vec8 *a8   = (Is16vec8 *) a;
  Is16vec8 *a8_1 = (Is16vec8 *) a_1;
  Is16vec8 *a8_2 = (Is16vec8 *) a_2;
```

```
  Is16vec8 sum0, sum1, sum2, min0, min1;

  Is16vec8 *pi8 = (Is16vec8 *) pi;



  if (nStates == 0)
    return(0);

  int Pack8States = nStates >> 3;

////ERROR IF LESS THAN 8 STATES OR IF STATES ARE NOT A MULTIPLE OF 8////
  //assert(Pack8States == 0 || (nStates % 8) > 1)
  if (Pack8States == 0 || (nStates % 8) > 1)
    return(-1);
////////////////////////////////////////////////////////////////////////
  switch (Pack8States)
  {
  case 1:
    {
      //step1:  Initialization of 8 states
      //Optimization - removed loop
      d8_1st = sat_add(pi8[0], b8[0]);



      //step2:  Loop
      //Optimization Goal:  Keep all data in registers by:
      //  Optimization 1)  Use only one loop instead of inner & outer loop
      //  Optimization 2)  No need to combine data with 2nd load.
      for( i=1; i < obsLen; i++ )
      {
        b8 = (Is16vec8 *)(bProb[obsVect[i]]);

        ////////////// Do the 1st Packed 8 Distance /////////
        d8_1_1st =  (Is16vec8)_mm_srli_si128((I128vec1)d8_1st, 2);
        //d8_1_1st |= (Is16vec8)_mm_slli_si128((I128vec1)d8_2nd,14);
        d8_2_1st =  (Is16vec8)_mm_srli_si128((I128vec1)d8_1st ,4);
```

```
        //d8_2_1st |= (Is16vec8)_mm_slli_si128((I128vec1)d8_2nd,12);

        sum0_1st = sat_add(d8_1st,  a8[0]);
        sum1_1st = sat_add(d8_1_1st,a8_1[0]);
        sum2_1st = sat_add(d8_2_1st,a8_2[0]);

        min0_1st = simd_min(sum2_1st,simd_min(sum1_1st,sum0_1st));
        //store result of first packed 8 distance
        d8_1st = sat_add(b8[0],min0_1st);

      }


      //Step 3 of Viterbi Algorithm:  Termination
      // Get 8 Minimum Distances
      // Optimization:  No loop needed
      min0 = d8_1st;
      break;
    }


case 2:
  {
    //step1:  Initialization of 8 states
    //optimization:  removed loop
    d8_1st = sat_add(pi8[0], b8[0]);
    d8_2nd = sat_add(pi8[1], b8[1]);


    //step2:  Loop
    //Optimization Goal:  Keep all data in registers by:
    //  Optimization 1)  Use only one loop instead of inner & outer loop
    //  Optimization 2)  No need to combine data with 3rd load.

    for( i=1; i < obsLen; i++ )
    {
      b8 = (Is16vec8 *)(bProb[obsVect[i]]);
```

```
      ////////////// Do the 1st Packed 8 Distance /////////
      d8_1_1st =  (Is16vec8)_mm_srli_si128((I128vec1)d8_1st, 2);
      d8_1_1st |= (Is16vec8)_mm_slli_si128((I128vec1)d8_2nd,14);
      d8_2_1st =  (Is16vec8)_mm_srli_si128((I128vec1)d8_1st ,4);
      d8_2_1st |= (Is16vec8)_mm_slli_si128((I128vec1)d8_2nd,12);

      sum0_1st = sat_add(d8_1st,  a8[0]);
      sum1_1st = sat_add(d8_1_1st,a8_1[0]);
      sum2_1st = sat_add(d8_2_1st,a8_2[0]);

      min0_1st = simd_min(sum2_1st,simd_min(sum1_1st,sum0_1st));
      //store result of first packed 8 distance
      d8_1st = sat_add(b8[0],min0_1st);

      ////////////// Do the 2nd Packed 8 Distance /////////
      d8_1_2nd =  (Is16vec8)_mm_srli_si128((I128vec1)d8_2nd, 2);
      //d8_1_2nd |= (Is16vec8)_mm_slli_si128((I128vec1)d8_3rd,14);
      d8_2_2nd =  (Is16vec8)_mm_srli_si128((I128vec1)d8_2nd ,4);
      //d8_2_2nd |= (Is16vec8)_mm_slli_si128((I128vec1)d8_3rd,12);

      sum0_2nd = sat_add(d8_2nd,  a8[1]);
      sum1_2nd = sat_add(d8_1_2nd,a8_1[1]);
      sum2_2nd = sat_add(d8_2_2nd,a8_2[1]);

      min0_2nd = simd_min(sum2_2nd,simd_min(sum1_2nd,sum0_2nd));
      //store result of first packed 8 distance
      d8_2nd = sat_add(b8[1],min0_2nd);
    }

    //Step 3 of Viterbi Algorithm:  Termination
    // Get 8 Minimum Distances
    min0 = simd_min(d8_1st, d8_2nd);
    break;
  }

  case 3:
  {
    //step1:  Initialization of 8 states
```

```
//optimization:  removed loop
d8_1st = sat_add(pi8[0], b8[0]);
d8_2nd = sat_add(pi8[1], b8[1]);
d8_3rd = sat_add(pi8[2], b8[2]);


//step2:  Loop
//Optimization Goal:  Attempt to keep most data in registers by:
//  Optimization 1)  Use only one loop instead of inner & outer loop
//  Optimization 2)  No need to combine data with 4th load.
for( i=1; i < obsLen; i++ )
{
  b8 = (Is16vec8 *)(bProb[obsVect[i]]);

  ////////////// Do the 1st Packed 8 Distance ////////
  d8_1_1st =  (Is16vec8)_mm_srli_si128((I128vec1)d8_1st, 2);
  d8_1_1st |= (Is16vec8)_mm_slli_si128((I128vec1)d8_2nd,14);
  d8_2_1st =  (Is16vec8)_mm_srli_si128((I128vec1)d8_1st ,4);
  d8_2_1st |= (Is16vec8)_mm_slli_si128((I128vec1)d8_2nd,12);

  sum0_1st = sat_add(d8_1st,  a8[0]);
  sum1_1st = sat_add(d8_1_1st,a8_1[0]);
  sum2_1st = sat_add(d8_2_1st,a8_2[0]);

  min0_1st = simd_min(sum2_1st,simd_min(sum1_1st,sum0_1st));
  //store result of first packed 8 distance
  d8_1st = sat_add(b8[0],min0_1st);

  ////////////// Do the 2nd Packed 8 Distance ////////
  d8_1_2nd =  (Is16vec8)_mm_srli_si128((I128vec1)d8_2nd, 2);
  d8_1_2nd |= (Is16vec8)_mm_slli_si128((I128vec1)d8_3rd,14);
  d8_2_2nd =  (Is16vec8)_mm_srli_si128((I128vec1)d8_2nd ,4);
  d8_2_2nd |= (Is16vec8)_mm_slli_si128((I128vec1)d8_3rd,12);

  sum0_2nd = sat_add(d8_2nd,  a8[1]);
  sum1_2nd = sat_add(d8_1_2nd,a8_1[1]);
  sum2_2nd = sat_add(d8_2_2nd,a8_2[1]);
```

```
        min0_2nd = simd_min(sum2_2nd,simd_min(sum1_2nd,sum0_2nd));
        //store result of first packed 8 distance
        d8_2nd = sat_add(b8[1],min0_2nd);

        ////////////// Do the 3rd Packed 8 Distance /////////
        d8_1_3rd =  (Is16vec8)_mm_srli_si128((I128vec1)d8_3rd, 2);
        //d8_1_3rd |= (Is16vec8)_mm_slli_si128((I128vec1)d8_4th,14);
        d8_2_3rd =  (Is16vec8)_mm_srli_si128((I128vec1)d8_3rd ,4);
        //d8_2_3rd |= (Is16vec8)_mm_slli_si128((I128vec1)d8_4th,12);

        sum0_3rd = sat_add(d8_3rd,  a8[2]);
        sum1_3rd = sat_add(d8_1_3rd,a8_1[2]);
        sum2_3rd = sat_add(d8_2_3rd,a8_2[2]);

        min0_3rd = simd_min(sum2_3rd,simd_min(sum1_3rd,sum0_3rd));
        //store result of first packed 8 distance
        d8_3rd = sat_add(b8[2],min0_3rd);


      }

      //Step 3 of Viterbi Algorithm:  Termination
      // Get 8 Minimum Distances
      //optimization:  removed loop
      min0 = simd_min(d8_1st, d8_2nd);
      min0 = simd_min(min0, d8_3rd);

      break;
    }

default:
  {
    //Step 1 of Viterbi Algorithm:  Initialization
    for( i = 0; i < Pack8States; i++)
      {dist8[i] = sat_add(pi8[i],b8[i]);}

    //Step 2 of Viterbi Algorithm:  Recursion
    //Loop for each observation or obsCount
```

```
      for(i = 1; i < obsLen; i++)
      {
        b8         = (Is16vec8 *)(bProb[obsVect[i]]);

        //EightStateLoop
        for(int j = 0; j < Pack8States; j++)
        {
          d8 = dist8[j];
          d8next = dist8[j+1];

          //Remove unaligned loads by combining 2 aligned loads together
          d8_1 =  (Is16vec8)_mm_srli_si128((I128vec1)d8, 2);
          d8_1 |= (Is16vec8)_mm_slli_si128((I128vec1)d8next,14);
          d8_2 =  (Is16vec8)_mm_srli_si128((I128vec1)d8 ,4);
          d8_2 |= (Is16vec8)_mm_slli_si128((I128vec1)d8next,12);

          sum0 = sat_add(d8,  a8[j]);
          sum1 = sat_add(d8_1,a8_1[j]);
          sum2 = sat_add(d8_2,a8_2[j]);

          min0 = simd_min(sum2,simd_min(sum1,sum0));
          dist8[j] = sat_add(b8[j],min0);

        }

      }

      //Step 3 of Viterbi Algorithm:  Termination
      // Get 8 Minimum Distances
      min0 = dist8[0];
      for(i = 1; i < Pack8States; i++)
      {
        min0 = simd_min(min0, dist8[i]);
      }
      break;
    }
  }
```

```
    // Get Minimum Distance
    //min0 = m7 m6 m5 m4 m3 m2 m1 m0
    //min1 = xx xx xx xx m7 m6 m5 m4
    //min0 = xx xx xx xx m3 m2 m1 m0
    min1 = (Is16vec8) _mm_shuffle_epi32(min0, 0xE);
    min0 = simd_min(min0, min1);
    //min0 = xx xx xx xx m3 m2 m1 m0
    //min1 = xx xx xx xx xx xx m3 m2
    //min0 = xx xx xx xx xx xx m1 m0
    min1 = (Is16vec8)_mm_shufflelo_epi16(min0, 0xE);
    min0 = simd_min(min0, min1);
    //min0 = xx xx xx xx xx xx m1 m0
    //min1 = xx xx xx xx xx xx xx m1
    //min0 = xx xx xx xx xx xx xx min
    min1 = (Is16vec8)_mm_shufflelo_epi16(min0, 0x1);
    min0 = simd_min(min0, min1);
    //minDist = min
    minDist = _mm_extract_epi16(min0, 0);

    return(minDist);
}
```

Appendix A – Performance Data

Performance Data Revision History

Revision	Revision History	Date
2.0	Updated with 1.2 GHz Pentium 4 processor performance numbers	7/00
1.0	Original publication of document	9/99

Table 1: Performance Data of HMM Implementations

Performance Data in Microseconds		
Test Cases	**Pentium III Processor (733 MHz)**	**Pentium 4 Processor (1.2 GHz)**
SSE IVEC (General Optimizations)		
8 States	1.09	0.91
16 States	2.80	2.30
24 States	6.59	4.71
32 States	10.31	8.02
SSE IVEC (Specific Optimizations)		
8 States	0.63	0.52
16 States	2.41	1.69
24 States	5.99	4.45
32 States	9.99	7.74
SSE2 IVEC (General Optimizations)		
8 States	-	0.68
16 States	-	1.59
24 States	-	2.61
32 States	-	4.40
SSE2 IVEC (Specific Optimizations)		
8 States	-	0.41
16 States	-	1.16
24 States	-	2.07
32 States	-	4.41

Table 2: Speedups from Table 1 Performance Data

Implementations and Platforms	Speedup
Best SSE IVEC on Pentium 4 Processor vs. Best SSE IVEC on PentiumIII Processor	1.21 – 1.42
Pentium 4 Processor (Best SSE2 IVEC vs. Best SSE IVEC)	1.26 - 2.15
Best SSE2 IVEC on a Pentium 4 processor vs. Best SSE IVEC on a PentiumIII processor	1.53 – 2.80

Performance was measured using a 733 MHz Pentium III processor and a 1.2 GHz Pentium 4 processor. See Tables 3 and 4 in **Test Systems Configuration** on page A-3 for a detailed description of both systems. Tables 1 and 2, above, reflect warm cache performance measurements.

This app note discusses four implementations (plus a functional – non-optimized implementation) of the Viterbi algorithm. The IVEC implementations use the SIMD integer vector class (IVEC) of the Intel C/C++ Class Libraries for SIMD Operations. The PIVEC and PIVEC_GEN implementations use the SSE instructions. The WIVEC and WIVEC_GEN implementations use the SSE2 instructions. As discussed in the app note, the four implementations avoid the store-forwarding penalty by avoiding unaligned loads. The PIVEC_GEN and WIVEC_GEN only contain the optimization that avoids the store-forwarding penalty. The PIVEC and WIVEC implementations also include optimizations that depend on the number of HMM states. Notice that the implementations that optimize for the different HMM states are the best performers on both processors.

Table 2 summarizes the results from Table 1. Table 2 shows the speedup of the best SSE IVEC implementation on Pentium III processor vs. the same SSE IVEC implementation on Pentium 4 processor as a range: 1.21x – 1.42x. On the Pentium 4 processor the SSE2 vs. SSE speedup is 1.26 – 2.15x depending on the number of HMM states. This speedup is attributed to the larger SIMD width. By using SSE2, twice as many states can be handled per loop with fewer instructions. The speedup of the best SSE2 IVEC implementation on Pentium 4 processor vs. the best SSE IVEC implementation on Pentium III processor is 1.53x – 2.80x. The speedup takes into account both the micro-architecture speedup and the SSE2 speedup.

In summary, the SSE2 instructions provide a significant performance improvement to the Viterbi decoding algorithm. The increased SIMD width is the largest contributor to this speedup. Other optimizations include: using the `pminsw` instruction, avoiding the store-forwarding penalty, padding the vectors to avoid branches in the inner loop, and optimizing for the different HMM states.

Test Systems Configuration

Table 3: Pentium III Configuration

Processor	Pentium III Processor at 733 MHz
System	Intel® Desktop Board VC820
Bios Version	VC82010A.86A.0028.P10
Secondary Cache	256KB
Memory Size	128 MB RDRAM PC800-45
Ultra ATA Storage Driver	Production Candidate 6.00.012
Hard Disk	IBM DJNA-371800 ATA-66
Video Controller/Bus	Creative Labs 3D Blaster† Annihilator† Pro AGP nVidia GeForce256† DDR –32MB
Video Driver Revision	NVidia Reference Driver 5.22
Operating System	Windows† 2000 Build 2195

Table 4: Pentium 4 Configuration

Processor	Pentium 4 Processor at 1.2 GHz
System	Intel Desktop Board D850GB
Bios Version	GB85010A.86A.0014.D.0007201756
Secondary Cache	256KB
Memory Size	128 MB RDRAM PC800-45
Ultra ATA Storage Driver	Production Candidate 6.00.012
Hard Disk	IBM DJNA-371800 ATA-66
Video Controller/Bus	Creative Labs 3D Blaster Annihilator Pro AGP nVidia GeForce256 DDR –32MB
Video Driver Revision	NVidia Reference Driver 5.22
Operating System	Windows 2000 Build 2195